SPEK: A Storage Performance Evaluation Kernel Module for Block Level Storage Systems under Faulty Conditions

Xubin He, *Member, IEEE,* Ming Zhang, *Member, IEEE,* and Qing (Ken) Yang, *Senior Member, IEEE*

***Abstract*— This paper introduces a new benchmark tool, SPEK (Storage Performance Evaluation Kernel module), for evaluating the performance of block-level storage systems in the presence of faults as well as under normal operations. SPEK can work on both Direct Attached Storage (DAS) and block level networked storage systems such as storage area networks (SAN). Each SPEK consists of a controller, several workers, one or more probers, and several fault injection modules. Since it runs at kernel level and eliminates skews and overheads caused by file systems, SPEK is highly accurate and efficient. It allows a storage architect to generate configurable workloads to a system under test and to inject different faults into various system components such as network devices, storage devices, and controllers. Available performance measurements under different workloads and faulty conditions are dynamically collected and recorded in SPEK over a spectrum of time. To demonstrate its functionality, we apply SPEK to evaluate the performance of two direct attached storage systems and two typical SANs under Linux with different fault injections. Our experiments show that SPEK is highly efficient and accurate to measure performance for block-level storage systems.**

***Index Terms*— Measurement Techniques, Performance Analysis, Degraded performance, Data Storage, Disk I/O**

I. Introduction

BEING able to access data efficiently and reliably has become the first priority of many organizations in today's information age. To achieve this goal, a typical data storage system has built-in redundancies at various levels. At the storage device level, redundancy is achieved using RAID (redundant array of inexpensive disks) [1], [2]. At the controller level, multiple HBAs (host bus adapters) and NICs (network interface cards) are used. Redundant switches, bridges, and connecting cables are also employed at the network level. Software mechanisms are designed to bypass failed components to provide continued data availability. Different topological architectures and fault-tolerant mechanisms exist for a SAN (storage area network), and new ideas and technologies emerge rapidly [3]. In order to make design decisions and provide optimal storage solutions, it is highly desirable to have efficient benchmark tools to quantitatively evaluate performance of various SAN architectures under faulty conditions. Current benchmark tools focusing on performance evaluation are not efficient enough to accurately measure the behavior of storage systems. Under many circumstances, the performance of a storage system available to users is the result achieved by file systems. This result is influenced by many factors such as file system caches, data organization, and buffer caches, so it cannot represent the true performance of the storage. For some applications, such as databases, which can utilize the raw performance of a storage device directly, it is desirable and necessary to measure and compare different storage systems at the raw (block) level. It is also important for file system and OS designers to know how much potential raw performance they could exploit and how much optimization they have made.

Existing benchmark tools such as PostMark [4], IoZone [5], Bonnie++ [6], and IoMeter [7] are widely used to measure various storage systems. PostMark, IoZone, and Bonnie++ run at the file system level and therefore mainly characterize file system performance. Fig. 1 shows experimental performance measurements of a same SCSI disk under different file system options using PostMark, IoZone, and Bonnie++. Although we use the same disk and same measurement metric (throughput in terms of KB/second), these benchmark tools produce completely different performance results. Such deviations can be attributed to effects of the file system cache as well as different characteristics of file systems [8]. While IoMeter can run below file systems, its measured performance on Linux fluctuates dramatically due to the effects of buffer caches.

Many operating systems provide a "raw" interface

X. He is with the Department of Electrical and Computer Engineering,Tennessee Technological University, Cookeville, TN 38505.
E-mail: hexb@tntech.edu.
M. Zhang and Q. Yang are with the Department of Electrical and Computer Engineering, University of Rhode Island, Kingston, RI 02881.
Email: {mingz,qyang}@ele.uri.edu.

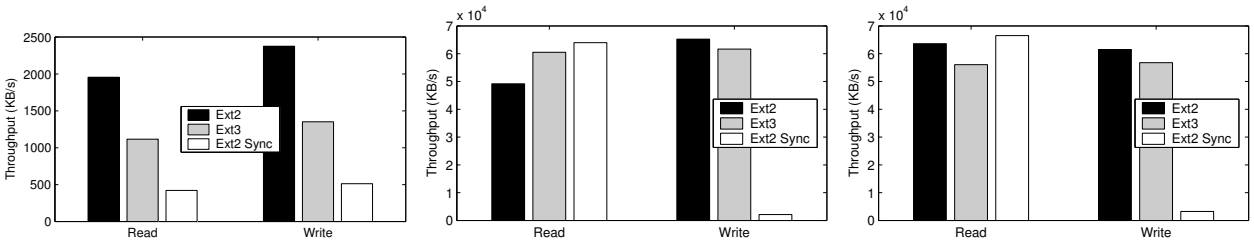


Fig. 1. Measured throughput of PostMark (left), IoZone (middle), and Bonnie++ (right). Although using the same hardware, measured performance change dramatically with changes of file system options.

bypassing the file system. Simply using such an interface will not efficiently produce accurate performance results for block level storage systems for the following two reasons. First, the raw interface provides a character device, as opposed to block device that is used by all storage systems. Second, since the "raw" interface is a user level interface, operating on it requires many context switches, giving rise to a great amount of overhead. Such overhead becomes significant when measuring high performance storage systems such as RAID and SAN.

Besides the accuracy problem of existing benchmark tools, they also raise an efficiency issue. Because these benchmarks run in a user space, excessive number of system calls and context switches result in large amounts of overhead. This problem is more pronounced when measuring high performance networked storage systems because the intensity of traffic generated by these benchmarks is limited due to excessive system overheads. As a result, a large number of clients are needed to saturate a high performance block level networked storage system.

In addition to performance measurement, degraded performance under faulty conditions is another concern for any storage system. Trivedi et al. [9]–[12] have extensively studied availability modeling for multi-processor systems and wireless communication networks using Markov reward models. Little work has been done on benchmark tools considering the degraded performance of a networked storage system under faulty conditions. One exception is the research by Brown and Patterson [13] who advocated for availability benchmarking with a case study on evaluating the availability of software RAID systems.

We introduce here a new benchmark tool for evaluating performance in consideration of various failure conditions of a storage system at the block level, which is referred to as SPEK (Storage Performance Evaluation Kernel module). The contributions of SPEK are two-fold. First, we propose a benchmark tool to a storage designer or purchaser to evaluate the performance of a storage system at the block level. The tool is highly efficient and accurate compared to existing benchmark tools that measure storage performance at the file system level. Second, our benchmark tool produces degraded performance measurements of storage architectures under faulty conditions. Specifically SPEK measures performance levels in the presence of various faults over time instead of using an average percentage of "up" time as an availability metric. It allows a user to generate workloads for a block level storage system, to inject faults at different parts of the storage system such as networks, storage devices, and storage controllers. By generating configurable workloads and injecting configurable faults, users can grab the dynamic changes of potentially compromised performance and therefore quantitatively evaluate system performance of a measured SAN. To demonstrate how SPEK works, we have performed several tests on direct-attached storage systems including single disks and disk arrays and two storage area network systems: an iSCSI-based [14] SAN and a STICS-based [15] SAN.

The paper is organized as follows. In next Section, we discuss the architecture and design of SPEK in detail. In Section 3, we present measurement results using SPEK on two direct attached storage systems and two networked storage systems. We discuss related work in Section 4 and conclude this paper in Section 5.

II. Architecture and Design of SPEK

The overall structure of SPEK is shown in Fig. 2. It consists of several components, including one SPEK controller, several SPEK workers, one or more SPEK probers, and different types of fault injection modules. A SPEK controller resides on a controller machine which is used to coordinate SPEK workers and probers. It can start/stop SPEK workers and probers, send commands, and receive responses from them. A Java GUI interface allows a user to input configuration parameters such as workload characteristics and to view measured results. Each SPEK controller also has a data analysis module to analyze measured data.

One SPEK worker runs on each testing client to generate storage requests via the low level device driver

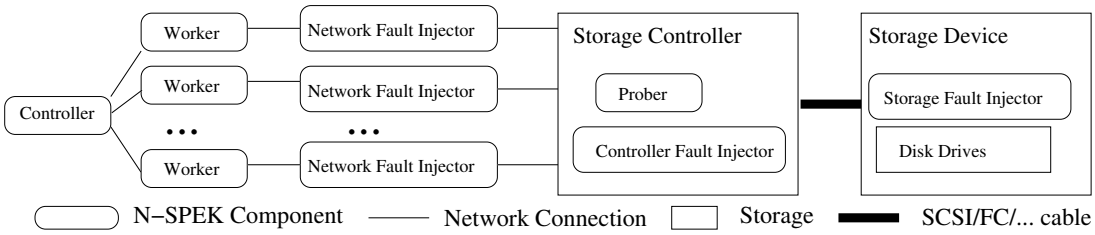


Fig. 2. SPEK Structure. It contains one SPEK controller, several SPEK workers, and one or several SPEK probers.

and to record performance data. A SPEK worker is a Linux kernel module running in the kernel space. Each SPEK worker has one main thread, one working thread, and one probe thread. The main thread is responsible for receiving instructions from the SPEK controller and controlling the working thread to execute the actual I/O operations. The working thread keeps sending storage requests to the SCSI layer, and these requests are eventually sent to remote targets by the lower level device driver. By using an event-driven architecture, SPEK is able to perform several outstanding SCSI requests concurrently, which is useful and necessary when testing SCSI tagged command features [16] and exploring the maximum throughput of a remote SCSI target. Many modern SCSI storage systems have the *command queue* feature that allows hosts to send several tagged commands and decide the specific execution sequence based on their own scheduling policies to get maximum overall throughput. The probe thread periodically records system status data and reports to the SPEK controller once a test completes. On each target device, a SPEK prober thread records system status for post-processing. Currently, we have developed a SPEK prober for Linux and plan to build SPEK probers for other platforms. Its functionality is similar to the probe thread in a SPEK worker.

To evaluate the degraded performance of a system, we need to inject faults at various parts of the system. Fault injection is commonly used in the fault-tolerance community to verify fault-tolerant systems or to study system behaviors [17]–[19]. It has also been adopted for the analysis of software RAID system availability [13] and measurement of networked service availability [20]–[22]. Three types of fault injection modules in SPEK support performance evaluation. By using these modules, users can introduce different types of faults to different parts of a networked storage system under test and measure the performance of the tested system at degraded modes. These modules are:

Network fault injector. It resides on a network bridge along the network path between a worker and the measured storage target. It injects unexpected events into network traffic traveling through the bridge by adding excessive delays and dropping packets with a configurable packet loss rate. Note that TCP provides reliable transport over the Internet through flow control, time-out, and retransmission mechanisms. Many network faults, including hardware and software failures, result in excessive delays at the transport layer; therefore,injecting excessive delays at TCP layer mimics various network faults. We call these faults *delay faults*. Our fault injector makes use of a program that controls the existing *dummynet* [23] package in FreeBSD, a network traffic control and shaping package previously used by other researchers [24].

Storage fault injector. This module generates some kinds of transient and sticky SCSI disk errors that may compromise the system performance and reliability. Our storage fault injection module is a RAM based virtual SCSI disk residing on the storage target. It exports itself as a normal SCSI disk and is utilized by the target under test. Previous researchers [13], [25] have also used disk emulation techniques to do fault injections and performance evaluation.

Controller fault injector. Besides hardware failures of a storage controller, major sources of faults of a controller can be attributed to malfunctions of the CPU and RAM. Normal operations of a controller can be compromised if required CPU and/or RAM resources are unavailable. Directed by configurable parameters, our controller fault injector takes most of the CPU and/or memory resources away from normal storage controller operations by adding unrelated CPU loads and memory loads to the controller.

A. Configuring Workload and Injecting Faults

The SPEK tool can take two types of workloads as input: realistic I/O traces and synthetic workloads. When it takes an I/O trace as input, SPEK replays it to the target under test. It can support different formats of I/O traces by implementing different converters. Currently, it supports trace formats from DTB [26] and SPC [27]. When it works under synthetic workload, the SPEK workload is generated by user configurable parameters

similar to those of IoMeter. Each SPEK worker generates workloads independently from each other allowing realistic networking environment to be simulated. The configurable parameters include:

- *Block size*. The current design of SPEK supports up to 8 different block sizes in one test run. A frequency weight is associated with each request block size representing how often a particular block size is used. For example, a sample workload may contain 10% of 8KB, 20% of 16KB, 30% of 32KB, and 40% of 64KB.
- *Number of transactions*. It controls how many transactions are generated in a test run. A transaction is defined as a block level read/write access.
- *Ramp up count*. This is a number used to bypass a transient period of the measurement process. Performance recording starts after the number of finished requests exceeds this number.
- *Burstiness*. It is defined as the length of a bursty request and the interval between two successive bursts. As a special case, when the interval is zero, SPEK sends requests continuously untill all requests are finished.
- *Read/write ratio* and *sequential/random ratio*. Each time when a SPEK worker generates a new I/O request, it needs to decide whether the request is a read or write and whether the address of this request will be continuous relative to the previous request (to be sequential) or will be random. These two ratios decide the probability used by the worker. For example, if the read ratio is 60% and the total number of requests is 10,000, then the total number of the read requests generated by the worker will approximate to 6,000.
- *Request address alignment*. It defines how a request address should align. Many storage systems perform quite differently when requests start from different addresses. For example, Linux performs best when the address of a request aligns to a 4K boundary. The default value is 512 Bytes.
- *Report time interval*. It defines the time interval for a SPEK worker to report performance data to a controller. By default,the interval is zero, which means a SPEK worker reports all data at the end of one test run. The interval ranges from zero seconds to one hour.

The fault injection modules in SPEK are also configurable by end users and can be set before a test experiment. Users can set them as sticky (steady-state) or transient [9], also known as permanent or intermittent faults [10]. A sticky (permanent) fault influences tested systems during the entire measurement period while a transient (intermittent) fault occurs in the system during a short period of time. For example, users can set a network delay fault to 1 ms during an entire test process as a sticky fault or add a packet loss rate of 0.0005 only in the third minute as a transient fault. By introducing these faults individually or simultaneously into a system, users can realistically simulate different failure situations and extract the performance of the measured system in a variety of circumstances.

B. Performance Metrics

SPEK mainly reports two performance values: throughput and response time. Throughput is represented in two forms: average I/O per second (IOPS) and average number of megabytes per second (MBPS). Response time includes average, minimum, and maximum response times. Users also have access to raw data to obtain medians and other statistics easily. During each test run, SPEK collects data related to performance and system status. There are two options to record and transfer such data to a SPEK controller: periodically at run time or one time at the end of each run. Unlike many other benchmark tools that collect some statistical data and compute them on the fly, SPEK provides two options: (1) deferring computation/analysis while allowing more data to be collected or (2) computing/analyzing data on the fly. The former option requires more memory space because it provides more detailed data to analyze performance dynamics of measured targets and gives users the flexibility to process and analyze measured raw data. A user can trade off memory and flexibility when doing performance testing.

In addition to throughput and response time, SPEK also records other performance related system status values. These performance counters include CPU load counters such as CPU utilization, user time, system time, interrupts per second, and context switches per second; network load counters including receive/send packets per second and receive/send bytes per second; and memory load counters such as free memory size, shared memory size, buffered memory size, swap size, swap exchange rate, and so forth. All these system status performance counters are recorded periodically with a user configurable interval.

III. Measurement Results

A. Experimental Setup

Several PCs are used in our experiments. All PCs have a single Pentium III 866 MHz CPU, 512 MB PC133 SDRAM and one or two Intel Pro1000 Gigabit NICs.

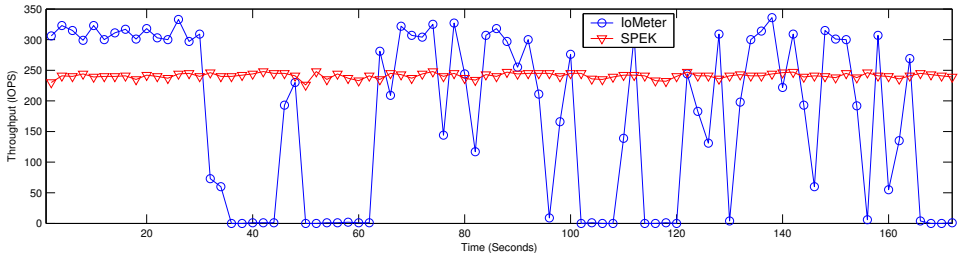


Fig. 3. Measured throughput for random write with block size being 16KB. The average IOPS of IoMeter is 175 while that of SPEK is 240. And the dynamic result of IoMeter fluctuates between 0 and 300 because of buffer cache effects while that of SPEK keeps consistent.

An Intel NetStructure 470T Gigabit Switch is used to connect all of them. All PCs run RedHat Linux 7.3 with a recompiled 2.4.18 kernel, except for one PC that acts as the bridge and runs FreeBSD 4.6. The iSCSI implementation comes from University of New Hampshire [14]. A Seagate SCSI disk (model: ST318452LW) is connected to the test target via an Adaptec 39160 Ultra 160 controller. The disk specifications are: Ulta160 SCSI interface, 18.4GB, 15000RPM, 2.0ms latency, and 4.2 ms average seek time.

In our performance experiments, one PC acts as a SPEK controller, three act as test clients, and one as a test target. In our experiments, four PCs act as a SPEK controller, a SPEK worker, a network bridge, and an iSCSI storage target, respectively. The network fault injection module is installed on the bridge, and the controller fault injection module resides on the iSCSI target. The iSCSI target uses an SPEK storage fault injection module and an emulated disk as a storage device. The SPEK worker is connected to the switch through the bridge using a crossover cable while the other three PCs are connected to the switch directly.

In order to verify the promises of SPEK, we have performed experiments to measure performance of Direct Attached Storage (DAS) systems as well as networked storage in consideration of faults using SPEK in comparison with existing benchmark tools. Most existing benchmark tools run at the file system level, with few exceptions such as IoMeter. We therefore compare our SPEK with IoMeter in terms of accuracy and efficiency.

B. Characterizing Direct Attached Storage Systems Using SPEK

In this section, we analyze the performance for a single disk and a simulated high performance disk array. Different experiments clearly show that our SPEK is suitable for measuring "raw" storage performance.

1) Measurements and Analysis of a Single Disk: In our first experiment, we measured the random write performance of a Seagate disk in terms of IOPS with each request size being 16KB as shown in Fig. 3. It is interesting to observe that the throughput produced by IoMeter fluctuates dramatically between 0 and 300 IOPS, while those produced by SPEK are fairly consistent over time. The fluctuations of the throughput produced by IoMeter result mainly from the buffer cache. Because of the existence of the buffer cache, throughput is high at times. However, Linux flushes the buffer cache when large enough sequential blocks are accumulated, every 30 seconds, or when dirty data exceeds a threshold value. Since our workload consists mainly of random writes, it is very unlikely to accumulate large sequential blocks. Most of flushing is caused by timeout and excessive dirty data. During a flushing period, measured throughput approaches zero because the system is busy and is unable to respond to normal I/O requests. This fact clearly indicates a limitation of IoMeter in accurately measuring disk I/O performance. Since our SPEK runs at a lower layer and is not affected by the buffer cache, as shown in Fig. 3, SPEK module, on the other hand, produces accurate and stable throughput values over time.

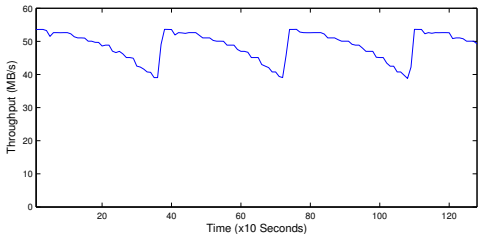


Fig. 4. Measured sequential read throughput on the Seagate disk using SPEK. SPEK correctly captures the ZCAV scheme of the Seagate disk.

The accuracy of SPEK is further evidenced by Fig. 4 that shows throughputs of the Seagate disk under sequential read workloads. In this figure, throughput changes periodically between 55MB/s and 39MB/s. We noticed that the total data accessed in each period is 18GB which is approximately the disk size. With Zoned Constant Angular Velocity (ZCAV) scheme [28], a modern SCSI disk has more sectors on the outer tracks than

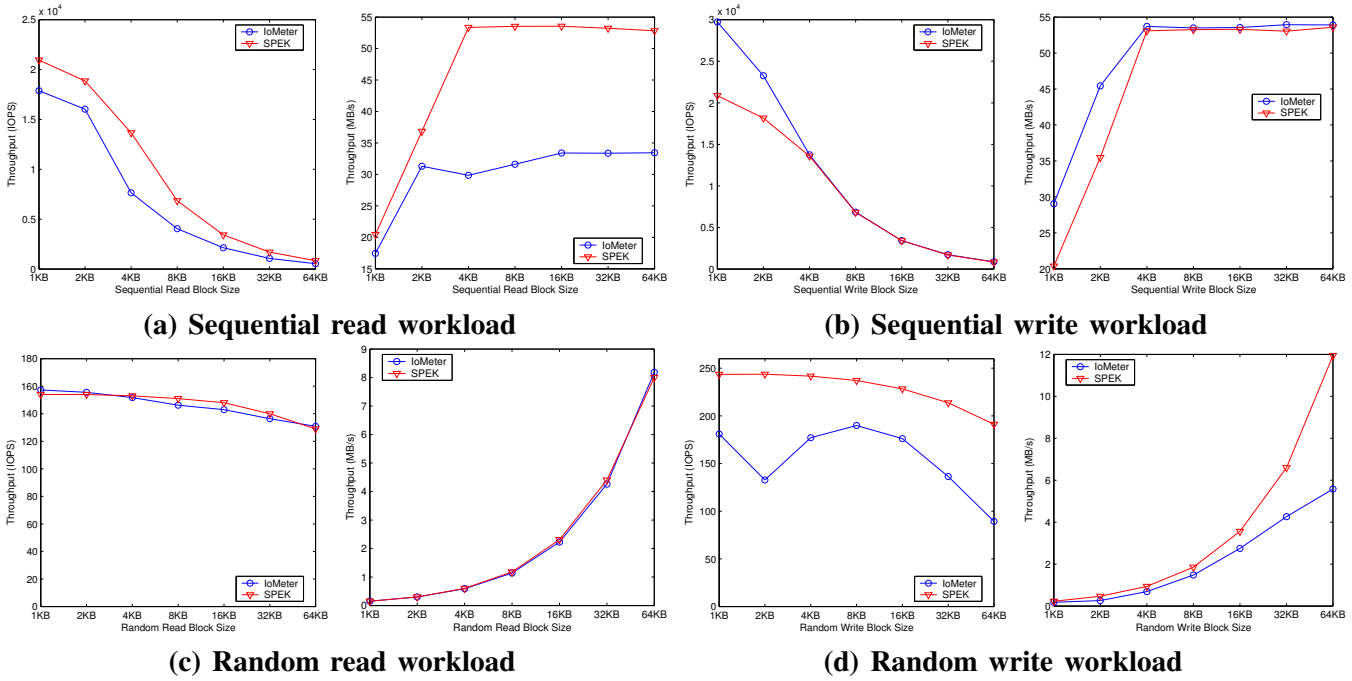


Fig. 5. Measurement results with different workloads on a Seagate disk (a,b,c,d).

the inner tracks. As a result, accessing sectors on the outer tracks is faster than the inner tracks, giving rise to a periodic throughput change, as shown in the figure. The sequential write throughput is almost identical to the read throughput; thus it is not shown here.

To provide a comprehensive comparison between IoMeter and SPEK, we performed experiments across different types of workload. The results are shown in Fig. 5. For sequential read workloads (Fig. 5a), IoMeter achieves lower throughput than SPEK. In terms of MBPS, throughput of IoMeter saturates at about 33MB/s while SPEK saturates at about 53MB/s. The difference results mainly from the system overheads for managing the file system cache and the buffer cache without providing any performance benefit because of sweeping data access. Note that for read operations, Linux will copy data read from the lower level to the file system cache for possible future reuse. For sequential writes, as shown in Fig. 5b, IoMeter produces better throughputs than SPEK for small request sizes. This is because written data bypass the file system cache, and the buffer cache collects small writes to form large sequential writes. As the request size increases, such differences diminish. All these measured data clearly indicate that throughputs produced by IoMeter are strongly influenced by the file system cache and the buffer cache. They do not accurately represent the actual performance of the underlying disk storage. On the other hand, SPEK accurately measures the raw performance of the block level storage devices. In the case of random read workloads, as shown in Fig. 5c, measured throughput by both IoMeter and SPEK are approximately equal. The reason is that the overheads due to the file system are negligible in this situation compared to tens of milliseconds caused by disk operations involving random seeks, rotation latencies, and transfers. Furthermore, the effect of the file system cache is also negligible, since 200,000 random read requests are uniformly distributed over the 18 GB disk space giving rise to approximately zero cache hit ratio. For random write workloads, as shown in Fig. 5d, the results are consistent with those in Fig. 3 with a 16KB block size for the same reasons explained previously. Note that Fig. 5d shows the average throughput, whereas Fig. 3 shows the throughput measured at a particular time instant.

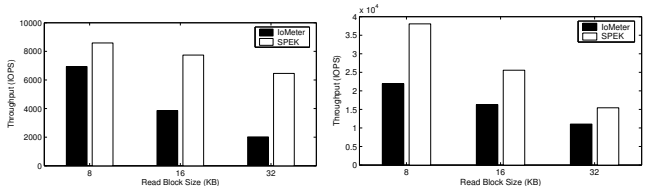


Fig. 6. IoMeter and SPEK measurement results on simulated storage with 100 μs average response time (a) and 10 μs average response time (b), respectively. SPEK generates more requests to saturate storage faster than IoMeter does because of lower overhead.

TABLE I
AVERAGE CONTEXT SWITCH NUMBERS PER I/O REQUEST OF IOMETER AND SPEK

	Sequential		Random	
	Read	Write	Read	Write
IoMeter	4.09	3.91	6.18	5.24
SPEK	1.98	2.01	2.03	2.02

2) Measurements and Analysis of a Disk Array: Our next experiment is to measure a RAID system to observe how SPEK and IoMeter perform in measuring high performance storage systems. In the absence of real high performance hardware and expensive storage systems, we built a SCSI disk device to emulate them using the Linux scsi_debug module. Basically the device is a RAM-based kernel space virtual storage system. Normal user-space applications can access this device from the user layer, similar to access any regular scsi disk drives. A SPEK worker can access it from the kernel layer directly. When the upper layer generates a read/write request to the virtual storage, the simulator simulates an access time that is user configurable. Fig. 6 shows the measurement results of the simulated high performance disk storages with disk access times 100 μs and 10 μs, respectively. As expected, file system overheads make the throughput measured using IoMeter to be much smaller than that measured using SPEK.

In addition to affecting the accuracy of performance measurements, file system overheads can also lower the efficiency of the measurements. Such low efficiency may require longer measurement times or more resources to carry out the same experiment. For example, if we were to measure the performance of an entry-level RAID system as a networked storage system as shown in Fig. 6a, two SPEK workers would be sufficient to saturate such a system using SPEK, while five workers would be necessary to saturate the storage using IoMeter. Readers may wonder how much file system overhead occurs using IoMeter as opposed to using SPEK. To give a quantitative view of such file system overheads, we measured the number of context switches as well as the number of system calls generated by the two benchmark tools. Table I lists the average number of context switches per I/O request with IoMeter and SPEK, respectively. The average numbers of context switches per I/O request generated by IoMeter and SPEK are 4.85 and 2.01, respectively. In terms of the number of system calls per I/O request, we found that an IoMeter worker generates about 14 system calls on average for each I/O request, while SPEK does not generate any system call because it is a kernel module. We used the HBench-OS [29] to measure context switches and system call overheads on our test clients. Each context switch cost ranges from 1.14 μs to 7.41 μs (average 4.27 μs) depending on the number of involved processes and the amount of context-related data. The costs of six typical system calls, including getpid, getrusage, gettimeofday, sbrk, sigaction, and write, are 0.352 μs, 0.579 μs, 0.517 μs, 0.036 μs, 0.696 μs, and 0.465 μs respectively, with an average cost of 0.440 μs. So for each I/O request, IoMeter has approximately 19 μs more overhead than SPEK, which is comparable with the average response time of a high-end RAID system, for example 10 μs for a RamSan-210 RAID system. This overhead hampers IoMeter when measuring a high-end storage system as verified by the results shown in Fig. 6. Thus, we believe that SPEK is especially efficient when measuring high performance storage systems. The context switching and system call overheads also explain why SPEK is superior to some benchmark tools that utilize the OS-provided raw access interface and run in user space.

C. Measurements and Analysis of Storage Area Networks Using SPEK

Previous measurements have shown that SPEK is very accurate and efficient to evaluate direct attached storage systems including both single disks and disk arrays. In this section we will show that SPEK also works well in measuring networked storage systems, especially storage area networks (SANs). We use SPEK to measure two typical SANs: an iSCSI-based SAN and a STICS-based SAN.

1) iSCSI and STICS Storage Area Networks: iSCSI is an emerging standard [30] to support remote storage access via encapsulating SCSI commands and data in IP packets. It was originally proposed by IBM, Cisco, HP, and others, and has recently become an industry standard approved by the IETF. It enables clients to discover and access SCSI devices directly via the mature TCP/IP technology and existing Ethernet infrastructures. Previous work on iSCSI mainly concentrated on performance evaluation and potential improvements [15], [24], [31], [32]. By using our SPEK, we have evaluated the degraded performance of a popular iSCSI implementation and observed that the performance of iSCSI degrades dramatically in case of faults, but it can rapidly recover after such faults are removed or corrected.

Using iSCSI to implement SAN over IP brings economy and convenience, however it also raises performance issues. We have recently proposed a new storage architecture: SCSI-To-IP Cache Storage, or STICS for short [15]. The purpose of STICS is to bridge the disparities

between SCSI and IP so that efficient SAN systems can be built over the Internet. Besides caching storage data, STICS also localizes SCSI commands and handshaking operations to reduce unnecessary traffic over the Internet. In this way, it acts as a storage filter to discard a fraction of the data that would otherwise move across the Internet, reducing the bottleneck problem imposed by limited Internet bandwidth. More information about STICS can be found in [15].

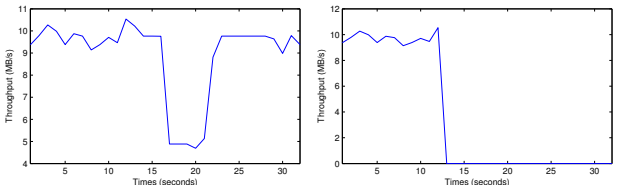
For both iSCSI-based SAN and STICS-based SAN, we have measured performance under different faults with different request patterns. We found that for different request patterns, the results show similar tendencies under the same faulty conditions, so we report here focusing on the measurements under sequential reads with an 8KB block size. In all experiments, we let a worker generate a next request only if it successfully receives a response for the previous request. If it gets a response indicating that the previous request failed, timed out, or finished with errors, it will retry the previous request.

2) Measurements of an iSCSI SAN under Synthetic Workloads: We have deployed an iSCSI SAN [14] environment for measurement purposes. We use DISKIO mode in the iSCSI target, allowing it to read/write Seagate disks. The iSCSI target exports several SCSI devices for test clients. We run different numbers of test clients using a sequential read workload with a 32KB block size. The results are shown in Table II. We found that the iSCSI target is saturated at 29.007 MB/s using two test clients. Since it is a software iSCSI implementation, the TCP/IP and iSCSI protocol overheads [15] are the main reason why the target saturates rapidly. The CPU utilization of the iSCSI target is consistently larger than 90% when using two test clients and approaches 100% when using three test clients. Most of the time is consumed on the iSCSI sending thread, since for these read operations the target needs to send data out to clients.

TABLE II
THROUGHPUT (MB/S) MEASUREMENT OF ISCSI SAN USING SPEK WITH SEQUENTIAL READ WORKLOAD AND BLOCK SIZE BEING 32KB

	Client 1	Client 2	Client 3	Target
Test 1	18.012	N/A	N/A	18.012
Test 2	15.488	13.519	N/A	29.007
Test 3	9.035	7.645	6.534	23.214

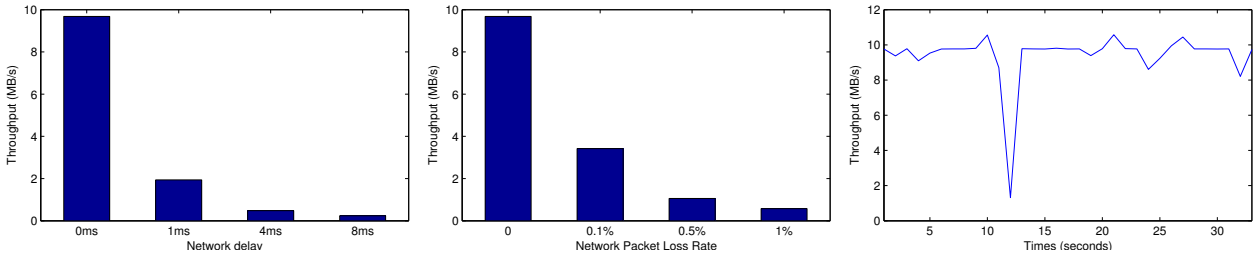
We plot the throughput results for iSCSI under different network faults in Fig. 7. From Fig. 7 (a) and (b), we observe that iSCSI performance degrades rapidly with increasingly severe delay faults and increasing packet loss rates. For example, when the network delay fault increases from 0 ms to 1 ms, the iSCSI performance drops from 9.68 MB/s to 1.94 MB/s, an 80% reduction. And a 0.001 packet loss rate will degrade the system performance from 9.68 MB/s to 3.42 MB/s. Since many iSCSI deployments share the same network with other applications, network congestion can greatly impair the performance of the iSCSI storage system. Fig. 7 (c) is the measured instant throughputs of the iSCSI target under transient packet loss sustained for about 2 seconds. During the 2 seconds, the system suffers from low throughput. In general, we find that the system runs with degraded performance during the time interval in which network faults are injected but returns to normal performance rapidly after the faults disappear.



(a) (b)

Fig. 8. iSCSI performance under (a) transient disk faults injected at time 15 seconds during 5 seconds and (b) sticky disk faults.

The iSCSI performance measurements under transient and sticky disk faults are shown in Fig. 8 (a) and (b), respectively. During the transient fault injection period, we let the storage fault injector reply to each I/O request with a successful response or with a correctable error response with the same probability. Such transient faults result in a nearly 50% performance drop during the period in which the transient errors are injected. To understand why there is such a big performance drop, we analyzed the source code of the iSCSI implementation. We noticed that in this iSCSI implementation, the storage controller simply returns the response of a request back to an initiator without checking the response. Therefore, for a failed request, the iSCSI controller sends the response containing an error message back to the client, and the client simply retries this request via the network. A better policy would be to let the iSCSI controller retry the failed request directly and return a successful or failed response after a predefined maximum number of trials. In this way, an iSCSI controller would handle most transient errors locally, minimizing unnecessary network traffic and thus improving iSCSI performance. With sticky uncorrectable disk errors injected, the iSCSI performance reduces to zero as shown in Fig. 8 (b). It can be seen that the iSCSI performance is greatly influenced

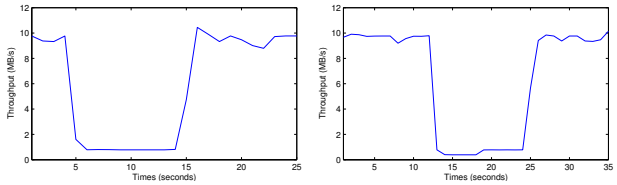


(a) (b) (c)

Fig. 7. iSCSI performance under several network faults. (a) Sticky delay faults. (b)packets loss faults injected. (b) Transient packets loss faults with 0.005 packet loss rate injected at time 10 during 2 seconds.

by the storage devices it uses. Storage devices such as RAID with some kind of redundancy or mirroring would have significantly greater availability and would enhance the performance at higher levels.

We have also measured the iSCSI performance under transient controller faults and show the results in Fig. 9 (a). During the fault-injection period the iSCSI only has 10% of its throughput in normal circumstances. This is because during a fault-injection period, the controller CPU becomes the bottleneck, although the Linux scheduler still gives the iSCSI process some time slices to run. Once the CPU fault is removed, iSCSI recovers its normal throughput rapidly.



(a) (b)

Fig. 9. iSCSI performance under (a) transient controller faults and (b) transient network and controller faults.

Fig. 9 (b) shows the performance results of iSCSI under multiple faults. With both network delay faults and controller CPU faults injected at time 10 for a duration of 5 seconds and 10 seconds respectively, iSCSI only gets a throughput of around 0.39 MB/s, much lower than in any any single fault scenario. After the network delay fault is removed at time 15, the performance of iSCSI recovers to around 0.79 MB/s, almost identical to the performance it achieves with a single controller CPU fault. Its performance recovers to the normal value rapidly after the CPU fault is removed.

3) Performance of an iSCSI SAN under a Commercial Workload: Besides the synthetic workloads, we also performed our tests with a commercial workload, an EMC trace, which was collected on an EMC Symmetrix disk array system installed at a telecommunication customer site. The trace file contains more than 230,000 storage requests with a fixed request size of 2KB. The trace is write-dominated, with 89% of operations being write operations. The average request rate is approximately 333 requests/second.

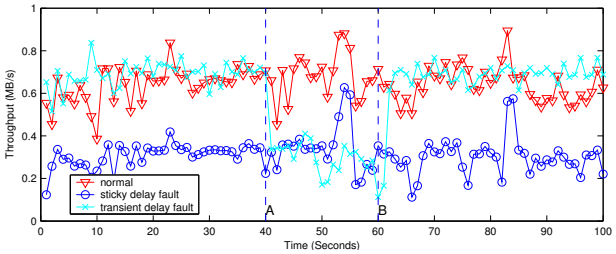


Fig. 10. iSCSI throughput under network delay faults.

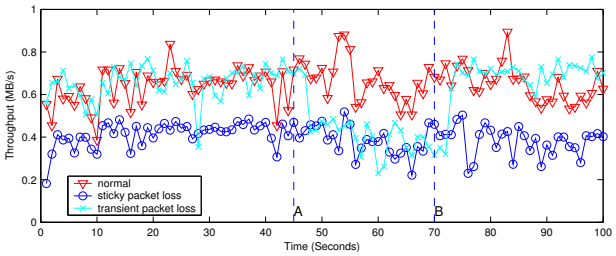


Fig. 11. iSCSI throughput under network packet loss faults.

Our measurement results for the iSCSI-based SAN are shown in Figures 10 through 11, and those for the STICS-based SAN are shown in Figures 14 and 15. Curves marked with triangles in these figures represent normal throughputs in terms of Megabytes per second without faults, those marked with "x" represent system throughputs under transient fault conditions, and those marked with circles represent throughputs under sticky fault conditions. Time point "A" marks the start of one or more types of transient faults being injected, and time point "B" marks the removal (or recovery) of the transient faults.

Figures 10 and 11 show throughput variations over time when network faults were injected at time point

A and removed at point B. For network delay faults, we added 4 ms delay to every packet going through the network bridge and for packet loss faults we set the packet loss rate to 1%. We notice in Figures 10 through 11 that when a network fault is injected, available system throughputs dropped by 50%.

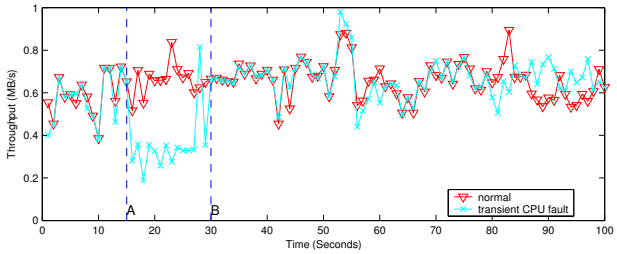


Fig. 12. iSCSI throughput under transient CPU faults.

Figure 12 shows available throughputs when transient controller faults are injected at time point A and removed at point B. A storage controller card hosts many codes such as a RAID control code, an iSCSI protocol stack, a TCP/IP stack, and so on, in addition to an on-board OS. A software bug may result in a temporary unavailability of the CPU to normal processes. We use our controller fault injector to emulate such faults by taking away 98% of the CPU resources. Such a CPU fault is injected at point A of Fig. 12, and results in an approximate 50% drop in throughput. The available throughputs gradually go back to normal after the faults are removed (at time point "B").

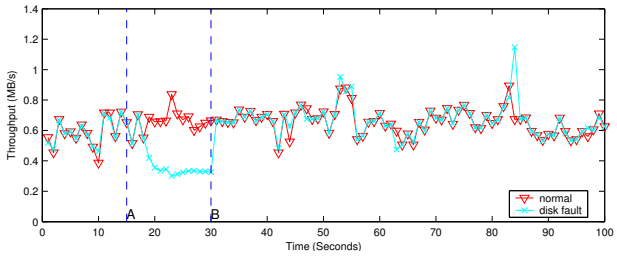


Fig. 13. iSCSI throughput under disk faults.

Impacts of disk faults are shown in Fig. 13. When a disk failure occurs, the simulated RAID system is assumed to automatically replace the faulty disk with a hot spare. The recovery process is done on-line, leading degraded throughput while recovery is taking place. Because the traffic intensity of the EMC trace is not as high, the recovery process is fairly quick, as shown in Fig. 13.

4) Performance of a STICS-based SAN: We also measured the performance of STICS-based SAN under network delay and loss faults. The results are shown in Figures 14 and 15. These two figures show throughput variations over time when network faults are injected for the EMC trace. Compared to the iSCSI-based SAN, the STICS-based SAN performs much better under those network faults.

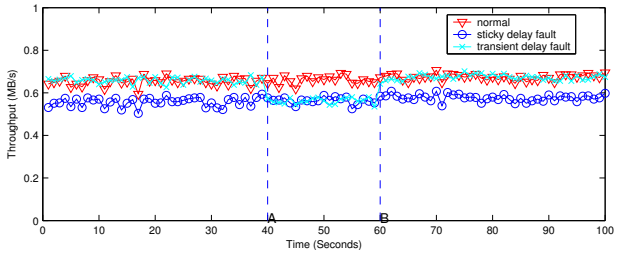


Fig. 14. STICS throughput under network delay faults.

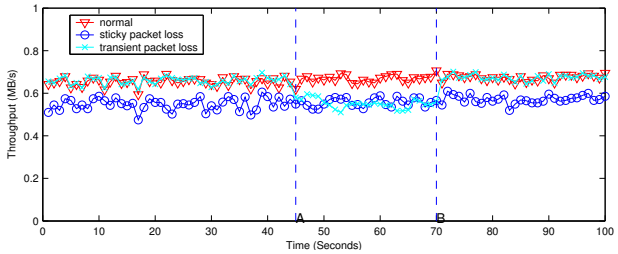


Fig. 15. STICS throughput under network packet loss faults.

IV. Related Work

Significant research has developed techniques and models to evaluate the performance, availability, and reliability in an integrated way over the past two decades [12], [33], [34]. The concept of performability [35], [36]captures the combined performance and dependability characters of the system. Since it is difficult to measure the performability directly, current system designers must rely on models [37]. Previous work on analytic performability estimation has concentrated on Markov reward models [38], [39]. Trivedi et al. have performed extensive analysis on composite performance and availability using stochastic reward Petri nets [11], [12], which are ultimately solved by translating them into Markov models.

The SPEK we have proposed in this paper is an effort to measure the performance of storage systems under faulty conditions. It is a benchmark tool to evaluate storage system performance efficiently.

Many I/O benchmark tools are available to measure I/O performance. Typical benchmark tools fall into three categories, as shown in Table III.

Most available I/O benchmark tools fall into the file system benchmarks category. Most of them create one or more files and perform read, write, append, and other

TABLE III
I/O BENCHMARK AND BENCHMARK TOOLS

Category	Benchmark Tools
File System Benchmark	Bonnie, Bonnie++, IoMeter, IoZone, LADDIS, NetBench, LMBench, VxBench, PostMark, SPEC SFS, IOGen, IOStone, IOBench, Pablo I/O Benchmark, NHT-1 I/O Benchmarks, NTIOgen
Standalone Disk I/O Benchmark	CORETest, Disktest, HD Tach, QBench, RawIO, SCSITool
Block level Networked Storage Benchmark	SPC-1, SPEK

operations on these files. Bonnie++ also has tests for file create, stat, and unlink operations. IOStone [40] only performs operations on a 1MB size file, making it impossible to get realistic results on modern storage systems because of their large file system caches. IOBench is obsolete and rarely used today. IoZone and IoMeter are the most popular among these benchmarks since they support many platforms and different file systems, including network file systems. IoZone is a file system benchmark allowing extensive file operations, including read, write, re-read, re-write, read backwards, read strided, fread, fwrite, random read, pread, mmap, aio_read, and aio_write. It reports throughput and response time results. IoMeter is originally from Intel and is now a sourceforge project. It is widely used and its workloads are highly parameterizable and configurable. While it claims to be a raw device test tool, IoMeter is still influenced by buffer caches under Linux, as evidenced in the previous section. LADDIS [41] and SPEC SFS [42] only operate on NFS, while NetBench [43] operates only on CIFS. PostMark [4] is also a widely used [25], [44] file system benchmark tool from Network Appliance. It measures performance in terms of transaction rates in an ephemeral small-file environment by creating a large pool of continually changing files. The Pablo I/O benchmark can be used to test MPI I/O performance, as well as application I/O, but still at the file system level. Its I/O Trace Library is very useful for analyzing application I/O behaviors while not aimed at block I/O measurement. NHT-1 I/O benchmark [45] measures application I/O, disk I/O, and network I/O, but its disk I/O measurement is still at the file system layer.

Many of the above mentioned benchmark tools perform well when used to measure file systems performance. IoMeter operates on the block device layer, bypassing the file system cache but is still affected by the buffer cache. There are also a few benchmark tools for measuring the block level or raw performance of storage devices. CORETest is a DOS disk benchmark tool from CORE International and is rarely used now. Disktest can be used to benchmark disk I/O performance, but its main purpose is to detect defects. Qbench is a DOS hard disk benchmark from Quantum Corporation that measures data access times and data transfer rates. SCSITOOL is a diagnostics and benchmarking tool for SCSI storage devices. The Pablo Physical I/O Characterization Tool [46], although not a benchmark tool, can be used to get useful trace information about disk I/O activity by using an instrumented disk device driver. There are also some research micro-benchmarks [47], [48]. Most of them are built to test some simple and limited I/O workloads, such as sequential read/write or random I/O workloads in fixed sizes and aimed at standalone storage systems.

None of the current benchmarks is able to measure performance of networked storage at a block level, an exception being SPC-1, which is focused in measuring block-level performance of networked storages [49]. SPC-1 is a standard specification being considered by the Storage Performance Council. It is not yet readily available to the public for performance evaluation purposes, although some incomplete performance data has been reported on the Web. In addition, SPC-1 has some limitations such as a maximum number of I/O streams and a lack of flexibility in defining each I/O stream [49]. To the best of our knowledge, our SPEK is the first benchmark tool for measuring block level performance of both DAS and networked storage systems with high flexibility, accuracy, and efficiency.

Our SPEK differs from the above tools mainly in three aspects. First, SPEK aims at measuring degraded performance for storage systems. Second, SPEK runs on the kernel level, bypassing the file system and reducing the overhead caused by system calls and context switches. Third, SPEK works well for both direct attached storage systems and block-level networked storage systems.

V. CONCLUSIONS AND FUTURE WORK

In this paper, we have proposed a new benchmark tool called SPEK (Storage Performance Evaluation Kernel module) for block level performance benchmarking of

storage systems. SPEK accurately measures the performance of a storage system in the presence of faults as well as under normal operations. It is able to benchmark both direct attached storage (DAS) and networked storage systems without the influence of the file system and low-level buffer caches. Performance results measured using our SPEK realistically represent the intrinsic performance of data storage systems. Users can easily configure SPEK to test a variety of workload scenarios and collect a variety of interesting performance metrics. Because it runs as a kernel module, system overheads such as system calls and context switches are minimized, making SPEK a highly efficient benchmarking tool. An early version of SPEK has been implemented to demonstrate its functionality and effectiveness and the tool including source code is available publically on the website at http://www.ece.tntech.edu/hexb/spek.tgz.

In the future, we plan to build a standard framework based on SPEK and integrate reliability measurements into this framework.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous referees for their insightful and constructive comments. They are also grateful to Martha Kosa for reading the paper and making suggestions and corrections. The first author's research is partially supported by the Research Office under a Faculty Research Grant and the Center for Manufacturing Research at Tennessee Technological University. The second and third authors' work has been supported in part by the US National Science Foundation under grants CCR-0073377 and CCR-0312613.

REFERENCES

[1] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson, "RAID : High-performance, reliable secondary storage," *ACM Computing Surveys*, vol. 26, no. 2, pp. 145–188, June 1994.

[2] M. Malhotra and K. Trivedi, "Reliability analysis of redundant arrays of inexpensive disks," *Journal of Parallel and Distributed Computing*, vol. 17, pp. 146–151, 1993.

[3] J. Ward, M. O'Sullivan, T. Shahoumian, and J. Wilkes, "Appia: Automatic storage area network fabric design," in *Proceedings of the Conference on File and Storage Technologies (FAST)*, Monterey, CA, Jan. 2002, pp. 203–217.

[4] J. Katcher, "PostMark: A new file system benchmark," Network Appliance, Tech. Rep. 3022, 1997.

[5] D. Capps and W. D. Norcott. Iozone filesystem benchmark. [Online]. Available: http://www.iozone.org/

[6] R. Coker. Bonnie++ benchmark tool. [Online]. Available: http://www.coker.com.au/bonnie++/

[7] Intel. Iometer, performance analysis tool. http://www.intel.com/design/servers/devtools/iometer/.

[8] K. A. Smith and M. I. Seltzer, "File system aging - increasing the relevance of file system benchmarks," in *Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, June 1997, pp. 203–213.

[9] A. Goyal, S. Lavenberg, and K. Trivedi, "Probabilistic modeling of computer system availability," *Annals of Operations Research*, vol. 8, 1987.

[10] O. Ibe, R. Howe, and K. Trivedi, "Approximate availability analysis of vaxcluster systems," *IEEE Transactions on Reliability*, vol. 38, no. 1, pp. 146–152, 1989.

[11] J. Muppala and K. Trivedi, "Composite performance and availability analysis using a hierarchy of stochastic reward nets," in *Proceedings of the 5th International Conference on Modeling Techniques and Tools for Computer Performance Evaluation*, Feb. 1991.

[12] Y. Ma, J. Han, and K. Trivedi, "Composite performance and availability analysis of wireless communication networks," *IEEE Transactions on Vehicular Technology*, vol. 50, no. 5, pp. 1216–1223, 2001.

[13] A. Brown and D. A. Patterson, "Towards availability benchmarks: A case study of software RAID systems," in *Proceedings of the 2000 USENIX Annual Technical Conference*, San Diego, CA, June 2000, pp. 263–276.

[14] UNH. iSCSI reference implementation. http://www.iol.unh.edu/consortiums/iscsi/.

[15] X. He, M. Zhang, and Q. Yang, "Stics:scsi-to-ip cache for storage area networks," *Journal of Parallel and Distributed Computing*, vol. 64, no. 9, pp. 1069–1085, 2004.

[16] *SCSI Block Commands*, NCITS Working Draft Proposed Standard, Rev. 8c, 1997. [Online]. Available: http://www.t10.org/scsi-3.htm

[17] J. Arlat, A. Costes, Y. Crouzet, J.-C. Laprie, and D. Powell, "Fault injection and dependability evaluation of fault-tolerant systems," *IEEE Transactions on Computers*, vol. 42, no. 8, pp. 913–923, 1993.

[18] S. Dawson, F. Jahanian, and T. Mitton, "ORCHESTRA: A fault injection environment for distributed systems," University of Michigan, Tech. Rep. CSE-TR-318-96, 1996.

[19] D. Pradhan, *Fault-tolerant Computer System Design*. Prentice Hall, 1996.

[20] X. Li, R. Martin, K. Nagaraja, T. Nguyen, and B. Zhang, "Mendosus: A SAN-based fault-injection test-bed for the construction of highly available network services," in *Proceedings of 1st Workshop on Novel Uses of System Area Networks (SAN-1)*, Feb. 2002.

[21] M. L. Shooman, *Reliability of Computer Systems and Networks: Fault Tolerance,Analysis, and Design*. John Wiley & Sons, 2002.

[22] K. Nagaraja, X. Li, R. Bianchini, R. Martin, and T. Nguyen, "Using fault injection and modeling to evaluate the performability of cluster-based services," in *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, 2003.

[23] L. Rizzo, "Dummynet: a simple approach to the evaluation of network protocols," *ACM Computer Communication Review*, vol. 27, no. 1, pp. 31–41, 1997.

[24] W. T. Ng, B. Hillyer, E. Shriver, E. Gabber, and B. Ozden, "Obtaining high performance for storage outsourcing," in *Proceedings of the Conference on File and Storage Technologies (FAST)*, Monterey, CA, Jan. 2002, pp. 145–158.

[25] J. L. Griffin, J. Schindler, S. W. Schlosser, J. S. Bucy, and G. R. Ganger, "Timing-accurate storage emulation," in *Proceedings of the Conference on File and Storage Technologies (FAST)*, Monterey, CA, Jan. 2002, pp. 75–88.

[26] Performance Evaluation Laboratory, Brigham

Young University. DTB: Linux Disk Trace Buffer. http://traces.byu.edu/new/Tools/.
[27] SPC. Storage Performance Council I/O traces. http://www.storageperformance.org/downloads.html.
[28] R. V. Meter, "Observing the effects of multi-zone disks," in *Proceedings of the 1997 USENIX Annual Technical Conference*, Anaheim, CA, Jan. 1997.
[29] A. Brown and M. Seltzer, "Operating system benchmarking in the wake of lmbench: A case study of the performance of netbsd on the intel x86 architecture," in *Proceedings of the 1997 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Seattle, USA, June 1997, pp. 214–224.
[30] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner. iSCSI draft standard. http://www.ietf.org/internet-drafts/draft-ietf-ips-iscsi-20.txt.
[31] K. Voruganti and P. Sarkar, "An analysis of three gigabit networking protocols for storage area networks," in *20th IEEE International Performance, Computing, and Communications Conference*, Phoenix, Arizona, Apr. 2001.
[32] K. Meth, "iSCSI initiator design and implementation experience," in *19th IEEE Symposium on Mass Storage Systems*, Adelphi, MD, Apr. 2002.
[33] A. Heddaya and A. Helal, "Reliability, availability,dependability and performability: A user-centered view," Boston University, Computer Science Department, Tech. Rep. BU-CS-97-011, Dec. 1996.
[34] K. Nagaraja, N. Krishnan, R. Bianchini, R. Martin, and T. Nguyen, "Evaluating the impact of communication architecture on the performability of cluster-based services," in *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA 9)*, Anaheim, CA, Feb. 2003.
[35] J. Meyer, "On evaluating the performability of degradable computing systems," *IEEE Transactions on Computers*, vol. C-29, no. 8, pp. 720–731, 1980.
[36] J. Meyer, "Performability: A retrospective and some pointers to the future," *Performance Evaluation*, vol. 14, no. 3-4, pp. 139–156, 1992.
[37] G. Alvarez, M. Uysal, and A. Merchant, "Efficient verification of performability guarantees," in *Proceedings of the 5th International Workshop on Performability Modeling of Computer and Communication Systems*, Sept. 2001.
[38] R. Smith, K. Trivedi, and A. Ramesh, "Performability analysis: Measures, and algorithm, and a case study," *IEEE Transactions on Computers*, vol. 37, no. 4, pp. 406–417, 1988.
[39] K. Trivedi, *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*. John Wiley & Sons, 2001.
[40] A. Park and J. C. Becker, "IOStone: a synthetic file system benchmark," *Computer Architecture News*, vol. 18, no. 2, pp. 45–52, June 1990.
[41] M. Wittle and B. E. Keith, "LADDIS: The next generation in NFS file server benchmarking," in *In USENIX Association Conference Proceedings '93*, April 1993.
[42] SPEC. SPEC SFS benchmark. http://www.spec.org/osg/sfs97/.
[43] VeriTest. Netbench file system benchamrk. http://www.etestinglabs.com/benchmarks/netbench/netbench.asp.
[44] K. Magoutis, S. Addetia, A. Fedorova, M. Seltzer, J. Chase, A. Gallatin, R. Kisley, R. Wickremesinghe, and E. Gabber, "Structure and performance of the direct access file system(DAFS)," in *Proceedings of USENIX 2002 Annual Technical Conference*, Monterey, CA, June 2002, pp. 1–14.
[45] R. Carter, B. Ciotti, S. Fineberg, and B. Nitzberg, "NHT-1 I/O benchmarks," NAS Systems Division, NASA Ames, Tech. Rep. RND-92-016, Nov 1992.
[46] H. Simitci and D. A. Reed, "A comparison of logical and physical parallel I/O patterns," *The International Journal of Supercomputer Applications and High Performance Computing*, vol. 12, no. 3, pp. 364–380, Fall 1998.
[47] Y. Zhu and Y. Hu, "Can large disk built-in caches really improve system performance?" University of Cincinnati, Tech. Rep. 259, 2002.
[48] E. Zadok and J. Nieh, "FiST: A language for stackable file systems," in *Proceedings of the 2000 USENIX Annual Technical Conference, San Diego, CA*, June 2000.
[49] SPC. SPC benchmark 1(SPC-1) specification. http://www.storageperformance.org/Specifications/SPC-1_v150.pdf.

Xubin He received the PhD degree in electrical engineering from the University of Rhode Island, USA, in 2002 and both the BS and MS degrees in computer science from the Huazhong University of Science and Technology, China, in 1995 and 1997, respectively. He is an assistant professor of electrical and computer engineering at the Tennessee Technological University. His research interests include computer architecture, storage systems, computer security, and performance evaluation. He received the Ralph E. Powe Junior Faculty Enhancement Award in 2004 and TTU Chapter Sigma Xi Research Award in 2005. He is a member of the IEEE Computer Society, Sigma Xi, and ASEE.

Ming Zhang received his PhD degree in electrical engineering from the University of Rhode Island, USA, in 2002 and both the BS and MS degrees in computer science from the Huazhong University of Science and Technology, China, in 1997 and 2000, respectively. His research interests include computer architecture, networked storage systems, benchmarking, and performance evaluation. He is a student member of the IEEE and ACM.

Qing (Ken) Yang received his B.Sc. in computer science from Huazhong University of Science and Technology, Wuhan, China, in 1982, the M.A.Sc. in electrical engineering from University of Toronto, Canada, in 1985, and the PhD degree in computer Engineering from the Center for Advanced Computer Studies, University of Louisiana at Lafayette, in 1988. Presently, he is a Distinguished Engineering Professor in the Department of Electrical and Computer Engineering at The University of Rhode Island where he has been a faculty member since 1988. His research interests include computer architectures, memory systems, disk I/O systems, networked data storages, parallel and distributed computing, performance evaluation, and local area networks. He is a senior member of the IEEE Computer Society and a member of the SIGARCH of the ACM.